ZOOM OUT AND SEE BETTER: SCALABLE MESSAGE TRACING FOR POST-SILICON SoC DEBUG

Debjit Pal and Shobha Vasudevan

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
University of Illinois at Urbana-Champaign

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 2017	

4. TITLE AND SUBTITLE
Zoom Out and See Better: Scalable Message Tracing for Post-Silicon SoC Debug

5. FUNDING NUMBERS
N/A

6. AUTHOR(S)
Debjit Pal and Shobha Vasudevan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1308 W. Main St., Urbana, IL, 61801-2307

8. PERFORMING ORGANIZATION REPORT NUMBER
UILU-ENG-17-2203

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
IBM, 11501 Burnet Rd., Austin, TX 78758

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

We present a method for selecting trace messages for post-silicon validation of System-on-Chip (SoC). Our message selection is guided by specifications of interacting flows in common user applications. In current practice, such messages are selected based on designer expertise. We formulate the problem as an optimization of mutual information gain and trace buffer utilization. Our approach scales to systems far beyond the capacity of current signal selection techniques. We achieve an average trace buffer utilization of 98.96% with an average flow specification coverage of 94.3% and an average bug localization to only 21.11% of the potential root causes in our large-scale debugging effort. We present efficacy of our selected messages in debugging and root cause analysis using five realistic case studies consisting of complex and subtle bugs from the OpenSPARC T2 processor.

14. SUBJECT TERMS
Post-silicon debug; Observability; Tracing hardware; OpenSPARC T2

15. NUMBER OF PAGES
10

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Zoom Out and See Better: Scalable Message Tracing for Post-Silicon SoC Debug

Debjit Pal, Shobha Vasudevan
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign, Urbana, IL USA
Email: {dpal2, shobhav}@illinois.edu

***Abstract*—We present a method for selecting trace messages for post-silicon validation of System-on-Chip (SoC). Our message selection is guided by specifications of interacting flows in common user applications. In current practice, such messages are selected based on designer expertise. We formulate the problem as an optimization of mutual information gain and trace buffer utilization. Our approach scales to systems far beyond the capacity of current signal selection techniques. We achieve an average trace buffer utilization of 98.96% with an average flow specification coverage of 94.3% and an average bug localization to only 21.11% of the potential root causes in our large-scale debugging effort. We present efficacy of our selected messages in debugging and root cause analysis using five realistic case studies consisting of complex and subtle bugs from the OpenSPARC T2 processor.**

I. Introduction

Post-silicon validation [5] is a crucial component of the validation of a modern System-on-Chip (SoC). It is performed under highly aggressive schedules and accounts for more than 50% of the validation cost [7], [13]. Hardware tracing in a Design-for-Debug (DfD) architecture comprises selection of a small set of hardware signals and routing of them to an observation point, such as an internal trace buffer. In current industrial practice, signal selection for hardware tracing is a creative activity, depending heavily on the insight and experience of the designer. A disciplined approach to hardware tracing will be critical to streamline SoC post-silicon validation, since the omission of a critical signal manifests only during post-silicon debug, when it is too late for a new silicon spin.

For modern SoC designs, debug and validation of "usage scenarios" involving interaction of various IP blocks are a highly critical target. For example, for a smartphone, browsing the web while playing music may exercise a communication sequence among the CPU, video controller, and network controller. The target in-field usage scenarios are typically documented during architecture development of an SoC. During post-silicon validation, the validator systematically exercises these scenarios to ensure that the fabricated system performs as expected. Hardware tracing is often the only DfD component available for debugging usage scenarios, since it requires comprehension of message communication from different IPs during system execution [11].

In this paper, we present an approach for observability selection designed to target validation of usage scenarios. Given a collection of system-level flows [1], [4] and their constituent messages, our method selects a subset of messages that are most valuable during debug. We model each usage scenario as an interleaving of flows. We select for observation the message combinations with maximal mutual information gain. We use heuristics for packing messages to maximize utilization of the trace buffer.

There has been significant research on automating post-silicon signal selection [2], [3], [6], [8]. Most of the proposed approaches rely on the State Restoration Ratio (SRR) to identify profitable signals for hardware tracing. However, SRR and related methods are *not* aware of usage scenarios or any other high-level functional intent.

Indeed, in our experiments on a USB controller design, we found that using existing signal selection techniques [2] and [6], only 6.67% and 26% of required interface messages across various design blocks could be reconstructed, respectively. In contrast, our method selected 100% of the messages required for debugging the USB usage scenarios. SRR-based algorithms typically select flip-flops for tracing, whereas our method selects interface signals of IPs for tracing. Further, the SRR-based algorithms suffer severely from scalability issues.

To show scalability and viability, we performed our experiments on a publicly available multicore SoC design, OpenSPARC T2 [10]. The design contains several heterogeneous IPs and reflects many complex design features of an industrial SoC design. We injected complex and subtle bugs, with each bug symptom taking several hundred observed messages (up to 457 messages) and several millions clock cycles (up to 21290999 clock cycles) to manifest. Our analysis shows that we can achieve up to 100% trace buffer utilization (average 98.96%) and up to 99.86% flow specification coverage (average 94.3%). Our messages are able to localize each bug to no more than 6.11% of the total paths that could be explored. Our selected messages helped eliminate up to 88.89% of potential root causes (average 78.89%) and localize to a small set of root causes. To our knowledge, this is one of the most complex SoC designs in the published literature on automated hardware tracing. We injected complex and subtle bugs in the OpenSPARC T2 [10] and present here the results of five such large-scale debugging case studies. We demonstrate the value of our selected messages in debugging and root cause analysis in our case studies.

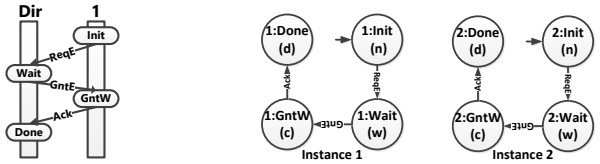
Our method relies on the availability of transaction-level

models. Time-to-market demands have caused a rapidly rising trend of developing transaction-level models as flows early in the SoC design cycle. Recent verification methodologies use these flows [1], [4], [9], [11]. We demonstrate the use of these flows in post-silicon observability instrumentation in this paper.

Our main contributions with this paper are threefold. First, we make scalability a prime focus of the post-silicon debug solution. In doing so, we operate at a higher level of abstraction (transaction level), as opposed to the RTL/gate-level signal tracing seen hitherto in the literature. Second, we exploit available architectural collateral (*e.g.*, messages, transaction flows) to develop targeted message selection for post-silicon hardware tracing. The system-level wide-angle view provides a tight link between the applications executed and the traced signals. Third, we provide a technique based on mutual information gain to select messages at the transaction level.

II. Preliminaries

Conventions. In SoC designs, a message can be viewed as an assignment of Boolean values to the interface signals of a hardware IP. In our formalization below, we leave the definition of the message implicit, but we will treat it as a pair $\langle \mathcal{C}, w \rangle$ where $w \in \mathbb{Z}^+$. Informally, $\mathcal{C}$ represents the content of the message, and w represents the number of bits required to represent $\mathcal{C}$. Given a message $m = \langle \mathcal{C}, w \rangle$, we will refer to w as the *bit-width of* m, denoted by *width*(m).



(a) (b)

Fig. 1: 1a shows a *flow* for an exclusive line access request for a toy cache coherence flow [11] along with participating IPs. 1b shows two legally indexed instances of cache coherence flow.

Definition 1: A *flow* is a directed acyclic graph (DAG) defined as a tuple, $\mathcal{F} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{S}_p, \mathcal{E}, \delta_f, Atom \rangle$ where $\mathcal{S}$ is the set of flow states, $\mathcal{S}_0 \subseteq \mathcal{S}$ is the set of initial states, $\mathcal{S}_p \subseteq \mathcal{S}$ and $\mathcal{S}_p \cap Atom = \emptyset$ is called the *set of stop states*, $\mathcal{E}$ is a set of messages, $\delta_{\mathcal{F}} \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S}$ is the transition relation and $Atom \subset \mathcal{S}$ is the set of atomic states of the flow.

We use $\mathcal{F}.\mathcal{S}, \mathcal{F}.\mathcal{E}$, etc. to denote the individual components of a flow $\mathcal{F}$. A *stop* state of a flow is its final state after its successful completion. The other components of $\mathcal{F}$ are self-explanatory. In Figure 1a, we show a toy cache coherence flow along with the participating IPs and the messages. In Figure 1a, $\mathcal{S}$ = {Init, Wait, GntW, Done}, $\mathcal{S}_0$ = {Init}, $\mathcal{S}_p$ = {Done}, and $Atom$ = {GntW}. Each of the messages in the cache coherence flow is 1 bit wide; hence $\mathcal{E} = \{\langle ReqE, 1 \rangle, \langle GntE, 1 \rangle, \langle Ack, 1 \rangle\}$.

Definition 2: Given a flow $\mathcal{F}$, an *execution* ρ is an alternating sequence of flow states and messages ending with a stop state. For flow $\mathcal{F}$, $\rho = s_0\ \alpha_1\ s_1\ \alpha_2\ s_2\ \alpha_3\ \ldots\ \alpha_n\ s_n$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}, \forall 0 \leq i < n,\ s_i \in \mathcal{F}.S,\ \alpha_{i+1} \in \mathcal{F}.\mathcal{E}$, and $s_n \in \mathcal{F}.\mathcal{S}_p$. The *trace* of an execution ρ is defined as $trace(\rho) = \alpha_1\ \alpha_2\ \alpha_3\ \ldots\ \alpha_n$.

An example of an execution of the cache coherence flow shown in Figure 1a would be ρ = {*n, ReqE, w, GntE, c, Ack, d*} and $trace(\rho)$ = {*ReqE, GntE, Ack*}.

Intuitively, one can see that a flow provides a pattern of system execution. A flow can be invoked several times, even concurrently, during a single run of the system. To make precise the relation between an execution of the system and the participating flows, we need to distinguish between instances of the same flow. The notion of *indexing* accomplishes that by augmenting a flow with an "index".

Definition 3: An *indexed message* is a pair $m = \langle \alpha, i \rangle$ where α is the message and $i \in \mathbb{N}$, referred to as the *index* of m. An *indexed state* is a pair $\hat{s} = \langle s, j \rangle$, where s is a flow state and $j \in \mathbb{N}$, referred to as the index of $\hat{s}$. An *indexed flow* $\langle f, k \rangle$ is a flow consisting of indexed message m and indexed state $\hat{s}$ indexed by $k \in \mathbb{N}$.

Figure 1b shows two instances of the cache coherence flow of Figure 1a indexed with their respective instance numbers. In our modeling, we ensure by construction that two different instances of the same flow do not have the same indices. Note that in practice, most SoC designs include architectural support to enable *tagging*, *i.e.*, unique identification of different concurrently executing instances of the same flow. Our formalization simply makes the notion of tagging explicit.

Definition 4: Any two indexed flows $\langle f, i \rangle$, $\langle g, j \rangle$ are said to be *legally indexed* either if $f \neq g$ or if $f = g$ then $i \neq j$.

Figure 1b shows two legally indexed instances of the cache coherence flow of Figure 1a. Indices uniquely identify each instance of the cache coherence flow.

A *usage scenario* is a pattern of frequently used applications. Each such pattern comprises multiple interleaved flows corresponding to communicating hardware IPs.

Definition 5: Let f, g be two legally indexed flows. The interleaving $f \,|||\, g$ is a flow called an *interleaved flow* defined as $\mathcal{U} = f \,|||\, g = \langle f.\mathcal{S} \times g.\mathcal{S}, f.\mathcal{S}_0 \times g.\mathcal{S}_0, f.\mathcal{S}_p \times g.\mathcal{S}_p, f.\mathcal{E} \cup g.\mathcal{E}, \delta_{\mathcal{U}}, f.Atom \cup g.Atom \rangle$, where $\delta_{\mathcal{U}}$ is defined as:

i) $\frac{s_1 \xrightarrow{\alpha} s_1' \ \wedge\ s_2 \notin g.Atom}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2 \rangle}$ and ii) $\frac{s_2 \xrightarrow{\beta} s_2' \ \wedge\ s_1 \notin f.Atom}{\langle s_1, s_2 \rangle \xrightarrow{\beta} \langle s_1, s_2' \rangle}$

where $s_1, s_1' \in f.\mathcal{S}$, $s_2, s_2' \in g.\mathcal{S}$, $\alpha \in f.\mathcal{E}$, $\beta \in g.\mathcal{E}$. Every path in the interleaved flow is an execution of $\mathcal{U}$ and represents an interleaving of the messages of the participating flows.

Rule i of $\delta_{\mathcal{U}}$ says that if s_1 evolves to the state s_1' when message α happens and g has a state s_2 that is not atomic/indivisible, then in the interleaved flow, if we have a state (s_1, s_2), it evolves to state (s_1', s_2) when message α happens. A similar explanation holds good for Rule ii of $\delta_{\mathcal{U}}$. For any two concurrently executing legally indexed flow f and g, $J = f \,|||\, g$, for any $s \in f.Atom$ and for any $s' \in g.Atom$,

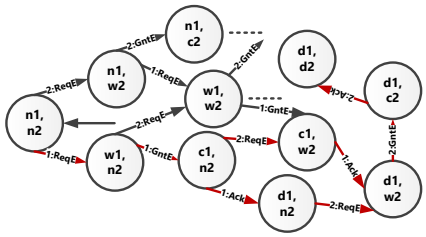


Fig. 2: Two instances of the cache coherence flow of Figure 1b interleaved.

$(s, s') \notin J.\mathcal{S}$. If one flow is in one of its atomic/indivisible states, then no other concurrently executing flow can be in its atomic/indivisible state.

Figure 2 shows a partial interleaving $\mathcal{U}$ of two legally indexed flow instances of Figure 1b. Since c_1 and c_2 both are atomic states, state (c_1, c_2) is an illegal state in the interleaved flow. $\delta_{\mathcal{U}}$ and the $Atom$ set make sure that such illegal states do not appear in the interleaved flows.

Trace buffer availability is measured in terms of bits thus rendering the bit width of a message important. In Definition 6, we define a message combination. Different instances of the same message i.e. indexed messages are not required while the bit width of the message combination is being computed.

Definition 6: A *message combination* $\mathcal{M}$ is an unordered set of messages. The *total bit width* W of a message combination $\mathcal{M}$ is the sum total of the bit widths of the individual messages contained in $\mathcal{M}$ i.e. $W(\mathcal{M}) = \sum_{i=1}^{k} width(m_i) = \sum_{i=1}^{k} w_i, m_i \in \mathcal{M}, k = |\mathcal{M}|$.

We introduce a metric called *flow specification coverage* to evaluate the quality of a *message combination*.

Definition 7: In a flow, every transition is labeled with a message. For a given message, the *visible state* is defined as the set of flow states reached on the corresponding transition. The *flow specification coverage of a message combination* is defined as the union of the visible flow states of all the messages, expressed as a fraction of the total number of flow states.

Mutual information gain measures the amount of information that can be obtained about one random variable by observing another. The concept of mutual information gain is heavily dependent on another probability theory concept, *entropy*. The mutual information gain of X relative to Y is given by $I(X;Y) = \sum_{x,y} p(x,y)\ log\frac{p(x,y)}{p(x)p(y)}$, where $p(x)$ and $p(y)$ are the associated probability mass function for two random variables X and Y respectively.

Maximizing information gain is done in order to increase flow specification coverage during post-silicon debug of usage scenarios. The message selection procedure considers the *message combination* $\mathcal{M}$ for tracing, whereas to calculate information gain over $\mathcal{U}$, it uses *indexed messages*.

Given a set of legally indexed participating flows of a usage scenario, bit widths of associated messages, and a trace buffer width constraint, **our method selects a message combination such that information gain is maximized over the interleaved flow** $\mathcal{U}$ and the **trace buffer is maximally utilized**.

III. Message selection methodology

For the cache coherence flow example of Figure 1a, we assume a trace buffer width of 2 bits and concurrent execution of two instances of the flow. *ReqE, GntE*, and *Ack* messages happen between *1-Dir, Dir-1*, and *1-Dir* IP pairs respectively. *ReqE, GntE*, and *Ack* consist of `req, gnt.` and `ack` IP signal and each of the messages is 1bit wide. $\mathcal{C}(ReqE) = \mathbb{B}^{|req|}$, $\mathcal{C}(GntE) = \mathbb{B}^{|gnt|}$, and $\mathcal{C}(Ack) = \mathbb{B}^{|ack|}$, $\mathbb{B} = \{0, 1\}$ denote the respective message contents.

A. Step 1: *Finding message combinations*

In Step 1, we identify all possible message combinations from the set of all messages of the participating flows in a usage scenario.

While we find different message combinations, we also calculate the total bit width of each such combinations. Any message combination that has a total bit width less than or equal to the available trace buffer width is kept for further analysis in Step 2[1]. Each such message combination is a potential candidate for tracing.

In the example of Figure 1a, there are 3 messages and $\sum_{k=1}^{3} \binom{3}{k} = 7$ different message combinations. Of these, only one (*ReqE, GntE, Ack*) has a bit width more than the trace buffer width (2). We retain the remaining six message combinations for further analysis in Step 2.

B. Step 2: *Selecting a message combination based on mutual information gain*

In this step, we compute the mutual information gain of the message combinations computed in step 1 over the interleaved flow. We then select the message combination that has the **highest mutual information gain** for tracing.

We use *mutual information gain* as a metric to evaluate the quality of the selected set of messages with respect to the interleaving of a set of flows. We associate two random variables with the interleaved flow namely X and Y_i. X represents the different states in the interleaved flow i.e. it can take any value in the set $\mathcal{S}$ of the different states of the interleaved flow. Let $\mathcal{M} = \bigcup_i \mathcal{E}_i$ be the set of all possible indexed messages in the interleaved flow. Let Y_i' be a candidate message combination and Y_i be a random variable representing all indexed messages corresponding to Y_i'. All values of X are equally probable since the interleaved flow can be in any state and hence $p_X(x) = \frac{1}{|\mathcal{S}|}$. To find the marginal distribution of Y_i, we count the number of occurrences of each indexed message in the set $\mathcal{M}'$ over the entire interleaved flow. We define $p_{Y_i}(y) = \frac{\text{\# of occurrences of } y \text{ in flow}}{\text{\# of occurrences of all indexed messages in flow}}$. To find the joint probability, we use the conditional probability and the marginal distribution i.e. $p(x,y) = p(x|y)p(y) = p(y|x)p(x)$.

[1]For multi-cycle messages, the number of bits that can be traced in a single cycle is considered to be the message bit width.

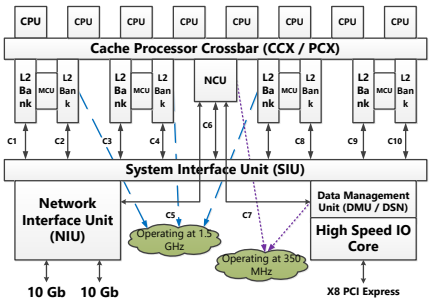


Fig. 3: Block diagram of OpenSPARC T2 processor [10]

$P(x|y)$ can be calculated as the fraction of the interleaved flow states x is reached after the message $Y_i = y$ has been observed. In other words, $p(x|y)$ is the fraction of times x that are reached, from the total number of occurrences of the indexed message y in the interleaved flow i.e. $p_{X|Y_i}(x|y) = \frac{\text{\# occurrence of } y \text{ in flow leading to x}}{\text{total \# occurrencess of } y \text{ in flow}}$. Now we substitute these values in $I(X;Y)$ to calculate the mutual information gain of the state set X w.r.t. Y_i.

In Figure 2, $p_X(x) = \frac{1}{15} \forall x \in \mathcal{S}$. Let $Y_1' = \{GntE, ReqE\}$ be a candidate message combination and Y_1 = {*1:GntE, 2:GntE, 1:ReqE, 2:ReqE*}. For $I(X;Y_1)$, we have $p(y = y_i) = \frac{3}{18}, \forall y_i \in Y_1$. Therefore, $p_{X|Y_1}(x|1 : GntE) = \{1/3 \text{ if } x = (c1, n2), 1/3 \text{ if } x = (c1, w2), 1/3 \text{ if } x = (c1, d2)\}$ and $p_{X,Y_1}(x, 1 : GntE) = \{1/18 \text{ if } x = (c1, n2), 1/18 \text{ if } x = (c1, w2), 1/18 \text{ if } x = (c1, d2)\}$.

Similarly, we calculate $p_{X,Y_1}(x, 2 : GntE)$, $p_{X,Y_1}(x, 1 : ReqE)$ and $p_{X,Y_1}(x, 2 : ReqE)$. The mutual information gain is given by $I(X, Y_1) = \sum_{x,y} p(x, y) log \frac{p(x,y)}{p(x)p(y)} = 1.073$.

Similarly, we calculate the mutual information gain for the remaining five message combinations. We then select the message combination that has the highest mutual information gain, which is $I(X, Y_1) = 1.073$, thereby selecting the message combination $Y_1' = \{$ReqE, GntE$\}$ for tracing. Intuitively, in an execution of $\mathcal{U}$ as shown in Figure 2, if the observed trace is {*1:ReqE, 1:GntE, 2:ReqE*}, immediately we can intuitively localize the execution to two paths shown in red in Figure 2 among the many possible paths of $\mathcal{U}$.

C. Step 3: *Packing the trace buffer*

Message combinations with the highest mutual information gain selected in Step 2 may not completely fill the trace buffer. To maximize trace buffer utilization, in this step we *pack* smaller message groups that are small enough to fit in the leftover trace buffer width. Usually, these smaller message groups are part of a larger message that cannot be fit into the trace buffer, *e.g.* in OpenSPARC T2, `dmusiidata` is a 20 bit-wide message whereas `cputhreadid` a subgroup of `dmusiidata` is 6 bits wide. We select a message group that can fit into the leftover trace buffer width, such that the information gain of the selected message combination in union with this smaller message group is maximal. We repeat this step until no more smaller message groups can be added in the leftover trace buffer. The enefits of packing are shown empirically in Section V-A.

TABLE I: Usage scenarios and participating flows in T2. **PIOR**: PIO read, **PIOW**: PIO write, **NCUU**: NCU upstream, **NCUD**: NCU downstream and **Mon**: Mondo interrupt flow. ✓indicates Scenario i executes a flow j and ✗ indicates Scenario i does not execute a flow j. Flows are annotated with (No. of flow states, No. of messages).

Usage Scenario	Participating Flows					Patici-pating IPs	Potential root causes
	PIOR (6, 5)	PIOW (3, 2)	NCUU (4, 3)	NCUD (3, 2)	Mon (6, 5)		
Scenario 1	✓	✓	✗	✗	✓	NCU, DMU, SIU	9
Scenario 2	✗	✗	✓	✓	✓	NCU, MCU, CCX	8
Scenario 3	✓	✓	✓	✓	✗	NCU, MCU, DMU, SIU	9

TABLE II: Simulation testbench details

Test Bench	Primary objective of testbench
tb1	Generate on-chip Mondo interrupt from PCI Express by injecting an error in memory management unit (MMU) of DMU, send PIO read and write request to IO
tb2	Generate on-chip Mondo interrupt using message signal interrupt (MSI), upstream and downstream memory request and NCU ASI register access
tb3	Upstream and downstream memory requests
tb4	Upstream and downstream memory requests, PIO read and write request to IO
tb5	Mondo interrupt generation in PCI express unit, PIO read and write request to IO
tb6	Mondo interrupt generation by sending a malformed MSI to the IMU of NCU, PIO read and write request to IO

In our running example, the trace buffer is filled up by the set of selected message combination. The flow specification coverage achieved with Y_1' is 0.7333.

IV. Experimental Setup

Design testbed: We primarily use the publicly available OpenSPARC T2 SoC [10] to demonstrate our result. Figure 3 shows an IP level block diagram of T2. Three different usage scenarios considered in our debugging case studies are shown in Table I along with participating flows (column 2–6) and participating IPs (column 7). We also use the USB design [12] to compare our approach with other methods that cannot scale to the T2.

Testbenches: We used 5 different tests from the `fc1_all_T2` regression environment. Details on the testbenches are shown in Table II. Each test exercises 2 or more IPs and associated flows. We monitored message communication across participating IPs during simulation and recorded the messages into an output trace file.

Bug injection: We created 5 different buggy versions of T2, which we analyze as five different case studies. Each case study comprises 5 different IPs. We injected a total of 14 different bugs across the 5 IPs in each case. Table III shows that the injected bugs are complex, subtle and realistic. Following [10] and Table III, we have identified several potential architectural causes that can cause an execution of a usage scenario to fail. Column 8 of Table I shows the number of potential root causes per usage scenario.

TABLE III: Representative bugs injected in IP blocks of OpenSPARC T2. *Bug depth* indicates the hierarchical depth of an IP block from the top. *Bug type* is the functional implication of a bug.

Bug ID	Bug depth	Bug category	Bug type	Buggy IP
1	4	Control	wrong command generation by data misinterpretation	DMU
2	4	Data	Data corruption by wrong address generation	DMU
3	3	Control	Wrong construction of Unit Control Block resulting in malformed request	DMU
4	4	Control	Generating wrong request due to incorrect decoding of request packet from CPU buffer	NCU
5	2	Control	Wrong request ID construction from memory controller to L2 cache	MCU
6	3	Control	Malformed vector for memory controller availability	NCU
7	3	Control	Misclassifying interrupt thereby generating wrong interrupt acknowledgement	NCU
8	3	Control	Selecting wrong FIFO to service interrupt request	CCX
9	4	Control	Wrong construction of clock domain crossing interrupt request packet	NCU
10	4	Data	Wrong qualification of uncorrectable error in peripheral data	NCU
11	3	Data	Generation of wrong address for read/write data	MCU
12	4	Control	Wrong encapsulation of function to tag and track packet transaction in internal data pipeline by wrong decoding of transaction type	DMU
13	5	Control	Wrong interrupt signal generation for PCI interrupts	DMU
14	3	Control	Wrong address generation to read data and also affecting read request validation	MCU

TABLE IV: Trace buffer utilization flow specification coverage and path localization of traced messages for 3 different usage scenarios. **FSP Cov**: Flow specification coverage (Definition 7), WP: With packing, WoP: Without packing. 32 bit-wide trace buffer assumed.

Case study	Usage Scenario	Trace Buffer Utilization		FSP Cov		Path Localization	
		WP	WoP	WP	WoP	WP	WoP
1	Scenario 1	96.88%	84.37%	99.86%	97.22%	0.13%	3.23%
2						0.31%	6.11%
3	Scenario 2	100%	71.87%	99.69%	93.75%	0.26%	5.13%
4						0.10%	2.47%
5	Scenario 3	100%	93.75%	83.33%	77.78%	0.11%	2.65%

TABLE V: Tracing statistics. **NoM**: Number of observed messages between sensitized bug location and observed symptom; **NoC**: Number of cycles between sensitized bug location and observed symptom.

Case study	Usage scenario	Symptom	NoM	NoC	Diagnosed buggy IP	Actual buggy IPs
1	Scenario 1	*FAIL: Bad Trap*	60	13647749	DMU	DMU, NCU
2			176	329250	NCU	NCU, CCX
3	Scenario 2	*FAIL: All Threads No Activity*	164	19701000	NCU	NCU, MCU
4			457	21290999	NCU	DMU, NCU
5	Scenario 3	*GLOBAL Time-Out*	65	18624749	MCU	MCU

TABLE VI: Comparison of signals selected by our method with those selected by SigSeT [2] and PRNet [6] for the USB design. **P**: Partial bit.

Signal Name	USB Module	Sig SeT	PR Net	Info Gain
rx_data	UTMI line speed	✗	✓	✓
rx_valid		✗	✓	✓
rx_active		✗	✓	✓
rx_err		✗	✓	✓
rx_data _valid	Packet decoder	✗	✗	✓
token _valid		✗	✗	✓
rx_data _done		✗	✗	✓
idma _done	Internal DMA	✓	✗	✓
tx_data	Packet assembler	✗	✗	✓
tx_valid		✗	✓	✓
tx_valid _last		✗	✗	✓
tx_first		✗	✗	✓
send_token	Protocol engine	✗	✗	✓
token_pid _sel		**P**	**P**	✓
data_pid _sel		**P**	✗	✓

V. Experimental Results

In this section, we provide insights into our large-scale effort to debug five different (buggy) case studies across 3 usage scenarios of the T2.

A. Flow specification coverage and trace buffer utilization

Table IV demonstrates the value of the traced messages with respect to flow specification coverage (Definition 7) and trace buffer utilization. These are the two objectives for which our message selection is optimized. Messages selected **without packing** achieve **up to 93.75% of trace buffer utilization** with **up to 97.22% flow specification coverage**. **With packing**, message selection achieves **up to 100% trace buffer utilization** and **up to 99.86% flow specification coverage**. This shows that we can cover most of the desired functionality while utilizing the trace buffer maximally.

B. Path localization during debug of traced messages

In this experiment, we use buggy executions and traced messages to show the extent of path localization per bug. Localization is calculated as the fraction of total paths of the interleaved flow. In Table IV, columns 7 and 8 show the extent of path localization.

We needed to explore **no more than 6.11% of interleaved flow paths** using our selected messages. With packing, we needed to explore **no more than 0.31% of the total interleaved flow paths** during debugging. Even with packing, subtle bugs like the NCU bug of buggy design 3 and buggy design 2 required exploration of more paths.

C. Statistics of messages traced for debugging

In Table V, for different bugs and cases, columns 4 and 5 show the number of messages traced and the number of cycles executed respectively between a sensitized bug location and the corresponding observed symptom. Empirically, it demonstrates the complexity and subtlety of the injected bugs.

D. Validity of information gain as message selection metric

We select messages per usage scenario. In Figure 4 we show the correlation between flow specification coverage and the mutual information gain of the selected messages. Flow

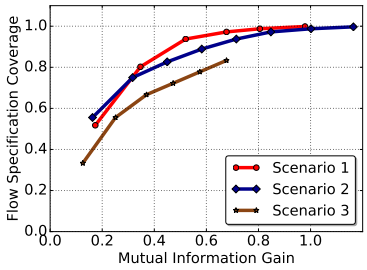


Fig. 4: Correlation analysis between *mutual information gain* and *flow specification coverage* for different message combinations for three different usage scenarios.

TABLE VII: Selection of important messages by our method

Message	Affecting Bug IDs	Bug coverage	Message importance	Selected	
				Y / N	Usage scenario
m1	8, 33, 36	0.21	4.76	Y	1, 2
m2	8, 33, 34, 36	0.28	3.57	Y	1, 2
m3	33, 36	0.14	7.14	Y	1, 2
m4	8, 29, 33	0.21	4.76	Y	1, 3
m5	18, 33	0.14	7.14	Y	1, 2
m6	-	-		N	-
m7	-	-		Y	1, 3
m8	33	0.07	14.28	Y	2
m9	1, 33	0.14	7.14	N	-
m10	24	0.07	14.28	Y	2
m11	1, 24	0.14	7.14	Y	2
m12	24	0.07	14.28	Y	2
m13	8	0.07	14.28	Y	2
m14	1, 17, 33	0.21	4.76	Y	2
m15	1, 17, 18, 33	0.28	3.57	N	-
m16	1, 17, 18, 33	0.28	3.57	Y	2, 3

specification coverage (Definition 7) **increases monotonically with the mutual information gain** over the interleaved flow of the corresponding usage scenario. This establishes that **increase in mutual information gain corresponds to higher coverage of flow specification**, indicating that mutual information gain is a good metric for message selection.

E. Comparison of our method to existing signal selection methods

To demonstrate that existing Register Transfer Level (RTL) signal selection methods cannot select messages in system-level flows, we compare our approach with an SRR-based method [2] and a PageRank-based method [6]. **We could not apply existing SRR based methods on the OpenSPARC T2, since these methods cannot scale. We instead use a smaller USB design for comparison with our method.**

In the USB [12] design we consider a usage scenario consisting of two flows: i) a flow for an 'OUT' packet that uses an internal DMA of the USB, and ii) a flow for an 'IN' packet that reads from the memory buffer. We apply our message selection algorithm on the interleaved flow of these two flows with a trace buffer width of 32 bits. To apply SigSeT and PageRank on netlists (PRNet), we synthesized the USB design and converted it to the ISCAS89 format. We compared traced and restored signals for both methods.

Table VI shows that our (mutual information gain based) method selects all of the `token_pid_sel`, `data_pid_sel` and other important interface signals for system-level debugging. SigSeT, on the other hand selects one interface signal completely and the other two interface bus signals partially. These are not useful for system-level debugging. Our messages are composed of interface signals, and achieve a flow specification coverage of **93.65%**, whereas messages composed of interface signals selected by SigSeT and PRNet have low flow specification coverage of **9%** and **23.80%** respectively.

F. Selection of important messages by our method

For evaluation purposes, we use *bug coverage* as a metric to determine which messages are important. A message is said to be *affected* by a bug if its value in an execution of the buggy design differs from its value in an execution of the bug-free design. If multiple bugs are affecting a message, we can see intuitively that it is highly likely that the message is a part of multiple design paths. The *bug coverage* of a message is defined as the total number of bugs that affect a message, expressed as a fraction of the total number of injected bugs. From the debugging perspective, a message is *important* if it is affected by very few bugs implying that the message symptomizes subtle bugs. Table VII confirms that post-Silicon bugs are subtle and tend to affect no more than 4 messages each. Column 4, 5 and 6 of Table VII show that our method was able to select important messages from the interleaved flow to debug subtle bugs.

Table VII shows that message *m15* is affected by four bugs and message *m9* is affected by two bugs, but because those messages are wider than the trace buffer size (32 bits), our method does not select them.

G. Effectiveness of selected messages in debugging usage scenarios

Every message originates from an IP and reaches a destination IP. Bugs are injected into specific IPs (see Table III). During debugging, sequences of IPs are explored from the point at which a bug symptom is observed in order to find the buggy IP. An IP pair (<source IP, destination IP>) is *legal* if a message is passed between them. We use the number of legal IP pairs investigated during debugging as a metric for selected messages. Table VIII shows that we investigated **an average of 54.67%** of the legal IP pairs, implying that our selected messages help us focus on a small percentage of the legal IP pairs.

To debug a buggy execution, we start with the traced message in which a bug symptom is observed and backtrack to other traced messages. The choice of which traced message to investigate is pseudo-random and guided by the participating flows.

Figure 5(a) plots the number of such investigated traced messages and the corresponding candidate legal IP pairs that are eliminated with each traced message. Figure 5(b) shows a similar relationship between the traced messages and the candidate root causes, *i.e.* the architecture level functions that might have caused the bug to manifest in the traced messages. Both graphs show that with more traced messages, more

TABLE VIII: Diagnosed root causes and debugging statistics for our case studies on OpenSPARC T2.

Case Study ID	Flows	Legal IP Pairs	Legal IP pairs investigated	Messages investigated	Root caused architecture level function
1	3	12	5	25	An interrupt was never generated by DMU because of wrong interrupt generation logic
2	3		6	67	Wrong interrupt decoding logic in NCU / corrupted interrupt handling table in NCU
3	3	10	8	142	Malformed CPU request from cache crossbar to NCU / erroneous CPU request decoding logic of NCU
4	3		6	199	Erroneous interrupt dequeue logic after interrupt was serviced
5	4	12	5	65	Erroneous decoding logic of CPU requests in memory controller

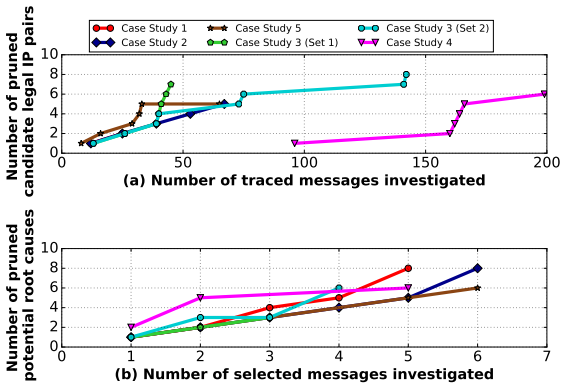
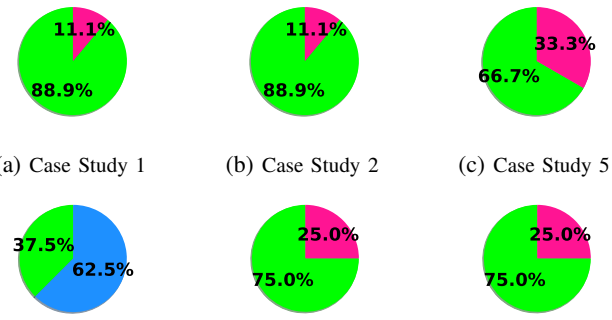


Fig. 5: Root causing buggy IP



(a) Case Study 1 (b) Case Study 2 (c) Case Study 5

(d) Case Study 3(Set 1) (e) Case Study 3(Set 2) (f) Case Study 4

Fig. 6: Selected messages-cause pruning distribution for diagnosis. Plausible Cause, Pruned Cause, Undecided Cause

candidate legal IP pairs as well as candidate root causes are progressively eliminated. This implies that every one of our traced messages contributes to the debug process.

Figure 6 shows that traced messages were able to prune out a large number of potential root causes in all five case studies. Our traced messages pruned out an **average of 78.89%** (**max. 88.89%**) of candidate root causes.

VI. Case Studies: Usage scenario debugging

It is illuminating to understand the debugging process for all of our case studies to appreciate the role of the selected messages.

For different bugs in different usage scenarios columns 3 and 4 of Table V show the number of messages traced and the number of cycles executed respectively between an observed bug symptom and the first suspicious message. This provides an insight into the complexity and subtlety of the bugs injected.

A. Case study 1

Symptom: In this experiment we used testbench *tb1* from Table II and traced messages from Table IX. The simulation failed with an error message *FAIL: Bad Trap*.

Debug with selected messages: We consider bug symptom causes of Scenario 1 (see Table IX) to debug this case. From the observed trace messages, `siincu` and `piowcrd`, we can see that the NCU got back a correct credit ID at the end of the PIO read and PIO write operations respectively. That rules out causes **8** and **9**. However, we cannot rule out causes **5**–**7** as we did not trace any messages related to PIO payload. A wrong payload may cause a computing thread to request an operand from the wrong memory location and thereby catch a *BAD Trap*. The absence of trace messages `mondoacknack` and `reqtot` implies that the NCU did not service any Mondo interrupt request and that the SIU did not request a Mondo payload transfer to the NCU respectively. Further, there is no message corresponding to `dmusiidata.cputhreadid` in the trace file, implying that the DMU was never able to generate a Mondo interrupt request for the NCU to process. This rules out all causes other than cause **4** (**1 cause out of 9, pruning 88.89% of the possible causes**) to explore further to find the root cause.

Root Cause: From the specification [10], we note that an interrupt is generated only when the DMU has credit and all previous DMA reads are done. We found no prior DMA read messages and the DMU had all its credit available. The absence of a `dmusiidata` message with CPUID = 0 and ThreadID = 0 implies that the DMU never generated a Mondo interrupt request. This makes the DMU a plausible location of the root cause of the bug.

B. Case study 2

In this experiment, we show that a set of IPs participating in a particular usage scenario and executing a set of flows may appear bug-free under one test bench but show erroneous behavior with another test bench. Despite this, our selected messages can help localize and identify the root cause of the incorrect execution.

Symptom: We used testbenches *tb5* and *tb6* (see Table II) and Scenario 1 messages (see Table IX) for this analysis. For *tb5* the simulation completed with message *PASS: Good*

TABLE IX: Possible root causes for Scenario 1

Selected Messages	Potential Causes	Potential Implication
reqtot, grant, mondoacknack, siincu, dmusiidata. cputhreadid, piowcrd,	**1.** Mondo request forwarded from DMU to SIU's bypass queue instead of ordered queue	**1.** Mondo interrupt not serviced
	2. Invalid Mondo payload forwarded to NCU from DMU via SIU	**2.** Interrupt assigned to wrong CPU ID and Thread ID
	3. Inappropriate handling of a Mondo Interrupt request by NCU	**3.** NCU stops servicing other interrupts
	4. Non-generation of Mondo interrupt by DMU	**4.** Computing thread fetches operand from wrong memory location
	5. Wrong PIO read request sent from NCU to DMU	**5.** DMU gets operand from wrong peripheral
	6. PIO read payload forwarded to SIUs ordered queue instead of bypass queue	**6.** Operand does not reach the computing thread
	7. Forwarding wrong PIO payload by DMU to NCU via SIU	**7.** Computing thread computes on wrong payload
	8. SIU returning wrong PIO read credit ID to NCU	**8.** NCU uses wrong credit ID in next PIO read operation
	9. DMU returning wrong PIO write credit ID to NCU	

End whereas for *tb6* the simulation terminated with an error message *FAIL: Bad Trap*.

Debug with selected messages: We consider bug symptom causes of Scenario 1 (see Table IX) to debug this case. From the observed trace messages, `siincu` and `piowcrd`, we can see that the NCU got back a correct credit ID at the end of the PIO read and PIO write operation respectively. That rules out cause **8** and **9**. However, we cannot rule out cause **5–7** as we did not trace any messages related to PIO payload. A wrong payload may cause a computing thread to request an operand from the wrong memory location and thereby catch a *BAD Trap*. The trace messages `reqtot` and `grant` confirm that a Mondo interrupt service request was forwarded from the SIU to the NCU and that the NCU granted SIU's request to transfer. That confirms that the Mondo request was placed in the SIU's ordered queue correctly thereby ruling out causes **1** and **4**. Tracing of the message `dmusiidata.cputhreadid` shows that a Mondo interrupt was assigned to the correct CPU ID and Thread ID, ruling out cause **2**. Thus it is necessary to explore the remaining trace messages `mondoacknack`. Although it was included in our trace message set, the trace file did not record any occurrence of `mondoacknack` implying that the NCU did not service the Mondo request successfully causing the computing thread to get into a *Bad Trap*. That immediately rules out causes **5–7** and points to cause **3** (**1 out of 9 causes, pruning 88.89% of the possible causes**) as the root cause.

Root Cause: This conflicting scenario of *tb5* and *tb6* requires architectural knowledge of how different interrupts are handled by different components of the NCU. After a Mondo request is delivered from the SIU to the NCU, the clock domain crossing `i2cbuf_siupio` splits it into control and data signals. `i2c_sctl` checks the Mondo status table to see if the current CPU ID and Thread ID are busy and responds with an `ack` or `nack` along with the ID. If the Mondo interrupt is accepted, the Mondo lookup and status tables are updated. Since in our case no `ack` or `nack` was generated, the problem could be in either `i2cbuf_siupio` or `i2c_sctl` or in the Mondo Table itself. It could be the case that in *tb5*, the NCU was able to identify the Mondo request but that in the case of *tb6* it failed to do so and dropped the packet altogether. It may also be the case that for *tb5* which generates the interrupt from the PCI express unit, the interrupt table was properly configured but that for *tb6* which generates the interrupt using the MSI, the interrupt table was corrupted. Once the problem has been localized to a subcomponent of the NCU, it is possible to use pre-silicon platforms to isolate the exact sub-component that is causing the issue.

C. Case study 3

Symptom: We use testbench *tb3* (see Table II) and Set 1 messages for Scenario 2 (see Table X) for this case analysis. The simulation fails with the error message *FAIL:All Threads No Activity GLOBAL TimeOut*.

Debug with selected messages: We consider Scenario 2 from Table X to debug this case. From the traced messages `reqtot`, `grant` and `mondoacknack` we can see that the Mondo interrupt was forwarded correctly from the SIU to the NCU, and that the NCU accepted the Mondo interrupt request, serviced it and sent an acknowledgement to DMU, ruling out causes **1–3**. Without those traced messages, it would have been impossible to understand the behavior of the Mondo interrupt during execution. We investigated messages `nc2cpx.ncucpxreq`, `mcu2ncu.rdackcpxnculd`, `pcx2ncu.idtxfr`, `ncu2mcu.rdreqpcx`, and `cpxncugnt` and found that the NCU received acknowledgement from the MCU via `mcu2ncu.rdackcpxnculd`. This raises a concern about the type of request received by the MCU from the NCU given that the testbench is designed to send *memory write requests* that are *non-posted i.e.* there should be no acknowledgement. This makes **10–14** plausible causes. None

TABLE X: Possible root causes for Scenario 2

Selected Messages	Potential Causes	Potential Implication
Set 1:nc2cpx.ncucpxreq, mcu2ncu.rdackcpxnculd, reqtot, cpxncugnt, grant, mondoacknack, pcx2ncu.idtxfr, nc2mcu.rdreqpcx	Causes **1.–3.** of Scenario 1 Table IX	Implications **1.–3.** of Scenario 1 Table IX
	10. Wrong encoding of the CPU request by PCX	**10.** PCX requests NCU to fetch wrong operand
	11. NCU forwards malformed CPU request to MCU	**11.** MCU fetches wrong operand from main memory
Set 2:pcx2ncu.idtfr, ncu2mcu.rdreqpcx, dmusiidata.ncucredid, reqtot, cpxncugnt, grant, mondoacknack, mcu2ncu.dpkt, ncu2mcu.dpkt	**12.** MCU decodes CPU requests wrongly	
	13. Wrong packetization of CPU operand by MCU to NCU	**13.** CPU gets wrong operand from NCU for computation
	14. NCU forwards malformed operand to CPX	

of those causes can be ruled out without knowing the type of request received by the MCU from the NCU. We did not trace the UCB `datapacket` (`ncu2mcu.dpkt`) from the NCU to the MCU, so couldn't get more insight.

To further debug this case, we changed the header packet definition from NCU to MCU and from MCU to NCU. The header packet definition now consisted of 1 bit of header valid and 4 bits of header data (a UCB packet) making the total header size 5 bits. This change led to the tracing of Set 2 messages from Scenario 2 in Table X. Rerunning the simulation allowed us to trace the `ncu2mcu.dpkt` from the NCU to the MCU which revealed that the NCU was sending *read requests* that ruled out causes **12**–**14** leaving causes **10** and **11** (**2 causes out of 8, a reduction of 75% of the possible causes**) as the only viable ones for further investigation.

Root Cause: Either a malformed CPU request from the PCX to the NCU or a wrong CPU request decoding logic in the NCU are the potential root causes for this bug symptom.

D. Case study 4

Symptom: We use testbench *tb2* (see Table II) and Scenario 2 Set 2 messages from Table X for this case study. The simulation failed with an error message *FAIL:All Threads No Activity GLOBAL TimeOut*.

Debug with selected messages: We consider bug symptom causes from Scenario 2 in Table X to debug this case. From the observed trace messages `ncu2mcu.dpkt` and `mcu2ncu.dpkt` we could quickly see that no malformed operand request was sent from the NCU to the MCU or from the MCU to the NCU. This implies that the CPU received a correct operand from main memory and rules out reason **10**–**14** for the bug symptom. That leaves reason **1**–**3** for further investigation. We trace four messages `dmusiidata.ncucredid`, `grant`, `reqtot`, and `mondoacknack` which were related to the Mondo interrupt. The *reqtot* and *grant* imply that the NCU accepted the Mondo interrupt request from the DMU via SIU, ruling out **1** because a Mondo interrupt request from the SIU's bypass queue will never reach the NCU. That leaves only causes **2** and **3** (**2 causes out of 8, pruning 75% of the possible causes**) for further investigation.

Root Cause: In the trace, via `mondoacknack`, we observed that `ack` and `nack` were both associated with the same `mondo id`. From the specification in [10] specification, a successfully serviced Mondo request should dequeue a serviced interrupt request from the ordered queue if the CPU and Thread ID are available, and move on to service the next request packet. However, `ack` and `nack's` association with the same Mondo id implies that after the Mondo interrupt was serviced and an `ack` was sent, the request was not dequeued from the SIU's ordered queue. When the NCU tried to service the next Mondo interrupt, it got the same request but the CPU ID and ThreadID for the request had already been allocated in the Mondo interrupt table. Hence, the NCU sent a `nack` but failed to dequeue the request again. That created a deadlock situation and stalled thread activity. This makes Mondo interrupt dequeue logic a potential candidate for the bug.

E. Case study 5

Symptom: We used testbench *tb4* (see Table II) and the set of messages from Scenario 3 in Table XI for this experiment. The simulation failed with an error message *FAIL:All Threads No Activity GLOBAL TimeOut*.

Debug with selected messages: We consider bug symptom causes from Scenario 3 in Table XI to debug this case. Trace messages `grant` and `siincu` together imply that a PIO read payload request was accepted successfully by the NCU and that the SIU returned correct NCU credit ID at the end of the PIO read operation. The trace message `piowcrd` shows that at the end of the PIO write, DMU sent back correct NCU credit ID to NCU. This rules out causes **16**–**18**. Since we did not trace any message related to the PIO read request between the NCU and the DMU, we cannot rule out cause **15**. The remaining set of trace messages show that the NCU received only 3 requests from the PCX. Of these 3 requests, one is an NCU ASI register access that was serviced by the NCU as shown by the `ncucpxreq` (contained in

TABLE XI: Possible root causes for Scenario 3

Selected Messages	Potential Causes	Potential Implication
grant, ncu2mcuload,	Causes **10.–14.** of Scenario 2 Table X	Implication **10.–14.** of Scenario 2 Table X
	15. Wrong PIO read request sent from NCU to DMU	**15.** DMU gets wrong operand from peripheral rendering computing thread inactive
piowcrd,	**16.** PIO read payload forwarded to SIUs ordered queue instead of bypass queue	**16.** Computing thread starve
mcu2ncu. rdackcpxnculd,	**17.** SIU returning wrong PIO read credit ID to NCU	**17.** NCU may use wrong credit ID in next PIO read operation fetching wrong operand for computing thread
cpxncugnt, siincu	**18.** DMU returning wrong PIO write credit ID to NCU	

ncu2mcuload) and cpxncugnt message in the trace file. The other two requests the NCU received from PCX comprise one read request and one write request. Using a 4 bit UCB datapacket (ncu2mcuload.dpkt) message, we could see that the NCU sent correct packets to the MCU with correct request, CPU and Thread IDs ruling out causes **10**, **11**, and **14** leaving causes **12**, **13** and **15** (**3 out of 9 causes, pruning 66.67% of the possible causes**) for more analysis.

Root Cause: As per the specification in [10] specification, i) on a successful read, the MCU sends an acknowledgement to the NCU via an rdackcpxnculd (part of mcu2ncu) message. Although rdackcpxnculd was included in our trace set, the execution of the Buggy design 5 does not record any occurrence of it in the trace file, implying that the MCU was never able to complete a read request and ruling out cause **15**. Since the NCU sent correct requests to the MCU yet the MCU failed to complete a read request successfully, the MCU is the plausible location of the root cause of the bug. Since the problem has been isolated to the MCU, it is possible to use pre-silicon platforms to isolate the exact sub-component of MCU that causes the issue.

VII. Conclusions

We have demonstrated the scalability and effectiveness of our trace message selection approach on the OpenSPARC T2 processor for root causing bugs in system-level usage scenarios. This is the largest-scale application of a hardware signal tracing approach in published literature.

References

[1] Y. Abarbanel, E. Singerman, and M. Y. Vardi. Validation of soc firmware-hardware flows: Challenges and solution directions. In *The 51st Annual DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 2:1–2:4, 2014.

[2] K. Basu and P. Mishra. Efficient trace signal selection for post silicon validation and debug. In *VLSI Design (VLSI Design), 2011 24th International Conference on*, pages 352–357. IEEE, 2011.

[3] D. Chatterjee, C. McCarter, and V. Bertacco. Simulation-based signal selection for state restoration in silicon debug. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pages 595–601. IEEE, 2011.

[4] R. Fraer, D. Keren, Z. Khasidashvili, A. Novakovsky, A. Puder, E. Singerman, E. Talmor, M. Y. Vardi, and J. Yang. From visual to logical formalisms for soc validation. In *Twelfth ACM/IEEE MEMOCODE 2014, Lausanne, Switzerland, October 19-21, 2014*, pages 165–174, 2014.

[5] T. Hong, Y. Li, S. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra. QED: quick error detection tests for effective post-silicon validation. In *2011 IEEE International Test Conference, ITC 2010, Austin, TX, USA, November 2-4, 2010*, pages 154–163, 2010.

[6] S. Ma, D. Pal, R. Jiang, S. Ray, and S. Vasudevan. Can't see the forest for the trees: State restoration's limitations in post-silicon trace signal selection. In *Proceedings of ICCAD 2015, Austin, TX, USA, November 2-6, 2015*, pages 1–8, 2015.

[7] P. Patra. On the Cusp of a Validation Wall. *IEEE Design and. Test of Computers*, 24(2):193–196, 2007.

[8] K. Rahmani, P. Mishra, and S. Ray. Efficient trace signal selection using augmentation and ILP techniques. In *Fifteenth ISQED 2014, Santa Clara, CA, USA, March 3-5, 2014*, pages 148–155, 2014.

[9] E. Singerman, Y. Abarbanel, and S. Baartmans. Transaction based pre-to-post silicon validation. In *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011*, pages 564–568, 2011.

[10] OpenSPARC T2, 2008. https://goo.gl/fRh1TK.

[11] M. Talupur, S. Ray, and J. Erickson. Transaction flows and executable models: Formalization and analysis of message passing protocols. In *FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 168–175, 2015.

[12] USB 2.0, 2008. http://opencores.org/project,usb.

[13] S. Yerramilli. Addressing Post-Silicon Validation Challenge: Leverage Validation and Test Synergy. In *Keynote, Intl. Test Conf.*, 2006.